# Algorithmic Problem Solving

## AAPS20 Strings II

Fredrik Präntare, Fredrik Heintz
Dept of Computer and Information Science
Linköping University

# Outline


- Suffix Trie
- Suffix Tree
- Suffix Array

# The Suffix



## suffix

/ˈsʌfɪks/

*noun*

1. a morpheme added at the end of a word to form a derivative (e.g. *-ation*, *-fy*, *-ing*, *-itis* ).
2. MATHEMATICS
   another term for subscript.

*verb*
/səˈfɪks/

1. append (something), especially as a suffix.

All **suffixes** of CARA:

CARA

ARA

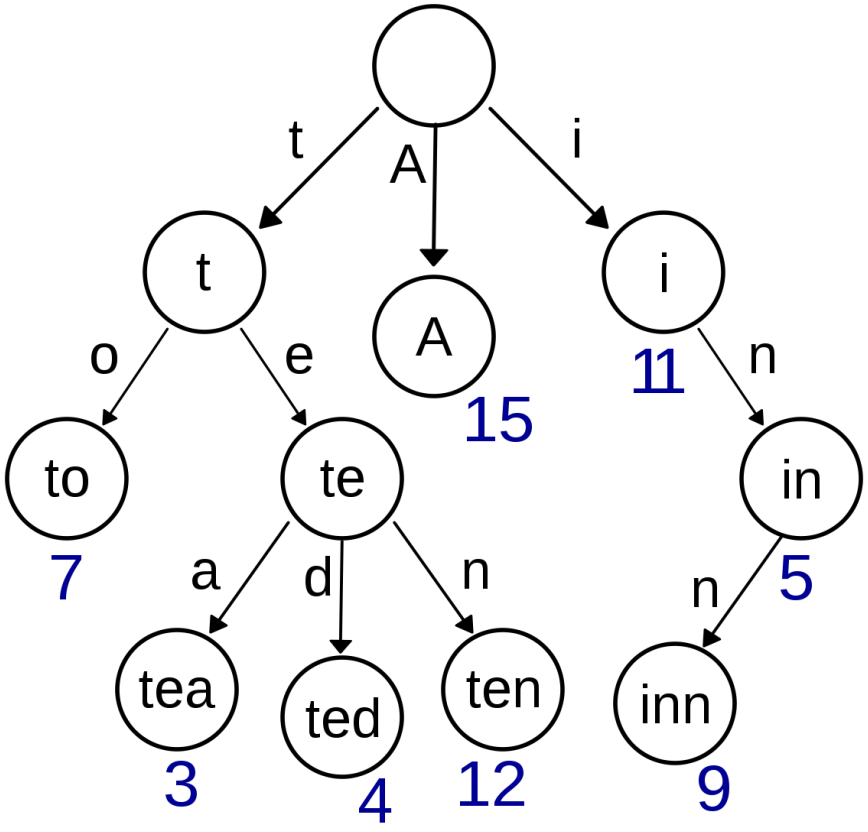
RA

A

# Trie



Recall that a ***trie*** (pronounced "try") is a tree that takes advantage of the structure of its keys (usually strings).



# Suffix Trie

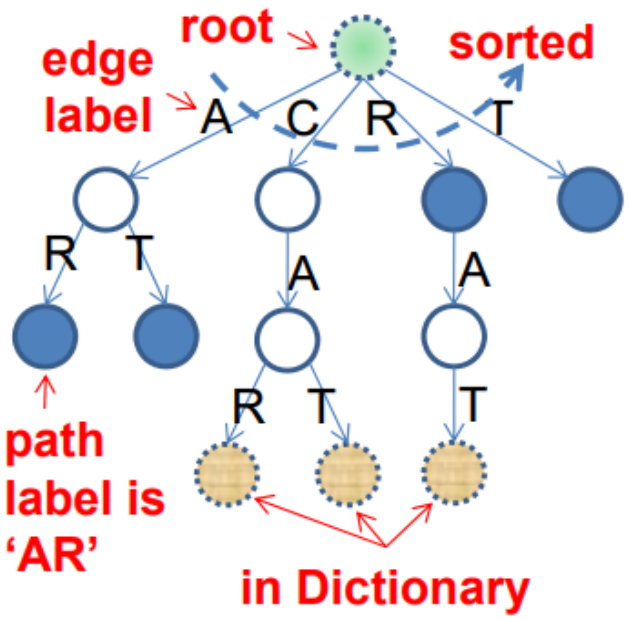


A ***suffix trie*** is a trie built for all suffixes of a set of strings.

**... with strings CAR, CAT and RAT:**

All Suffixes:
1. CAR
2. AR
3. R
4. CAT
5. AT
6. T
7. RAT
8. AT
9. T

Sorted Unique Suffixes:
1. AR
2. AT
3. CAR
4. CAT
5. R
6. RAT
7. T



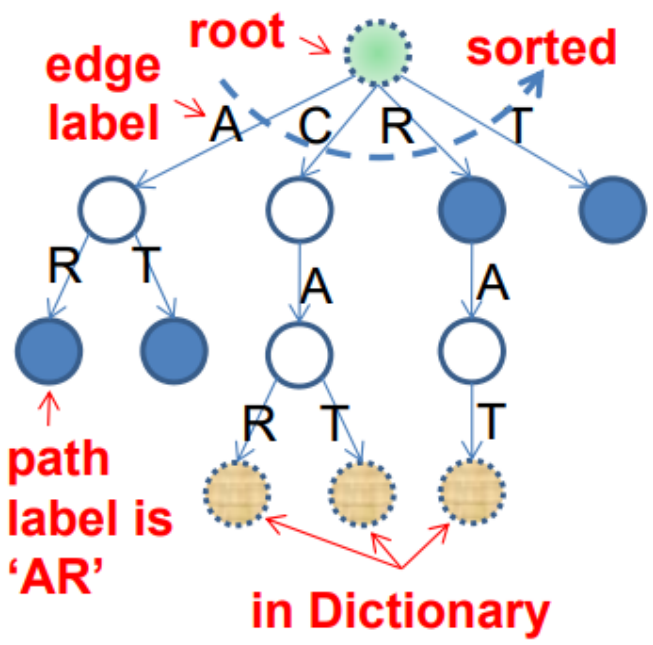

(example by Steven Halim)

# Suffix Trie



Example applications of a suffix trie given a string **S** and a trie **T**:

- **Check whether S is a substring of T.**
  *Follow the path for S from the root.*
  *If you exhaust S, then S is in T.*
- **Check whether S is a suffix of T.**
  *Follow the path for S from the root.*
  *If you end at a leaf at the end of S, then S is a suffix of T.*
- **Count # of occurrences of S in T.**
  *Follow the path for S from the root.*
  *The result is the sum of the # of suffixes represented by the leaves under the node you end up in.*



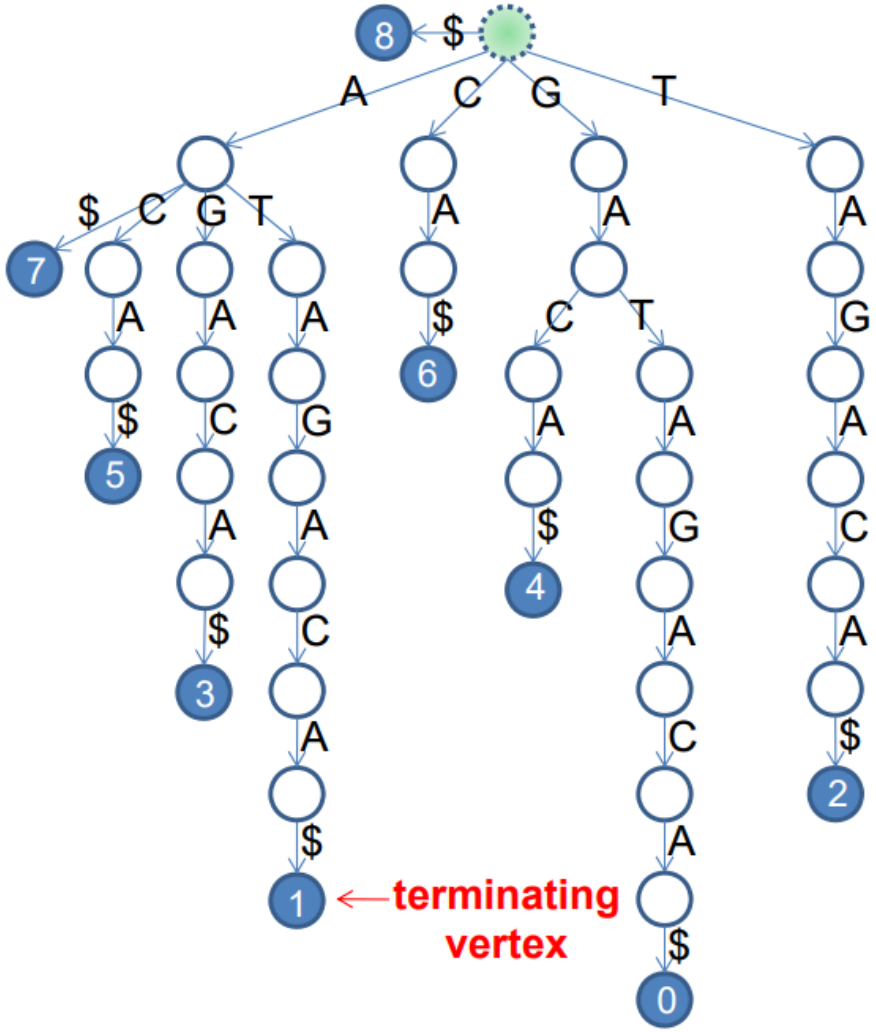

(example by Steven Halim)

# Suffix Trie



A suffix **trie** T = ”GATAGACA$”:

| i | Suffix |
|---|---|
| 0 | GATAGACA$ |
| 1 | ATAGACA$ |
| 2 | TAGACA$ |
| 3 | AGACA$ |
| 4 | GACA$ |
| 5 | ACA$ |
| 6 | CA$ |
| 7 | A$ |
| 8 | $ |



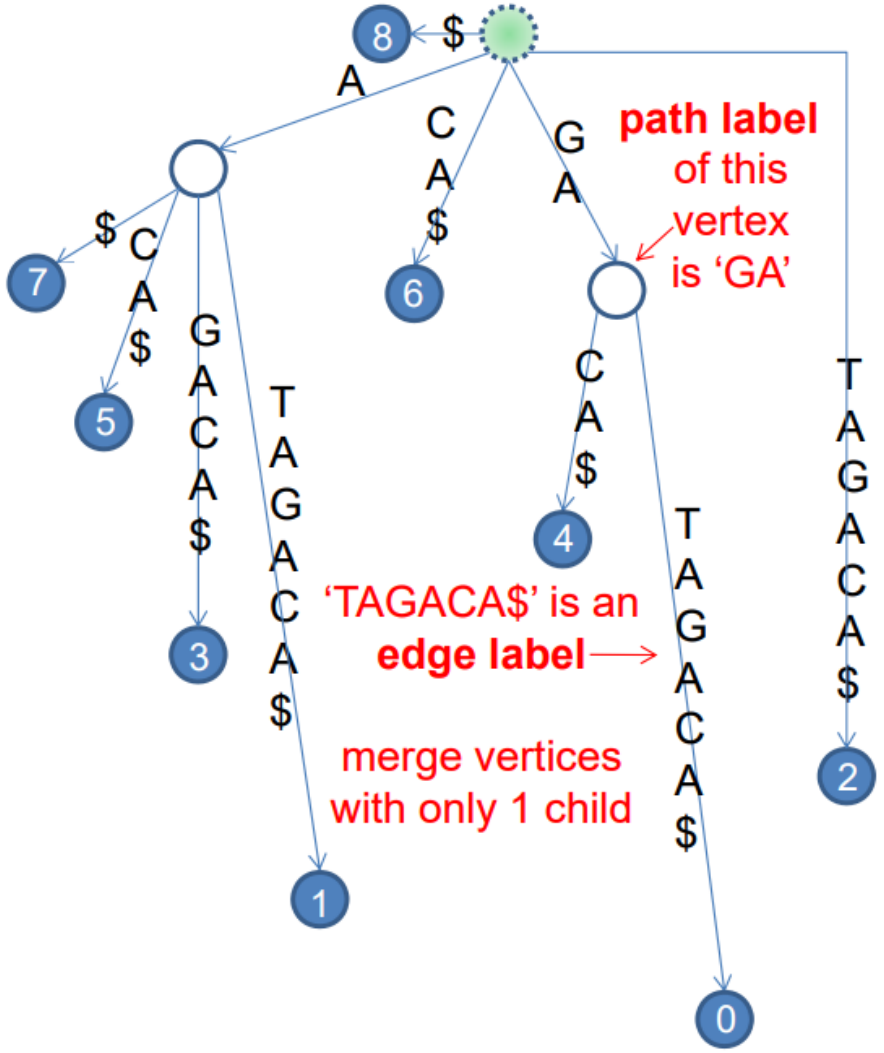

(example by Steven Halim)

# Suffix Tree



A suffix **tree** T = ”GATAGACA$”:

| i | Suffix |
|---|---|
| 0 | GATAGACA$ |
| 1 | ATAGACA$ |
| 2 | TAGACA$ |
| 3 | AGACA$ |
| 4 | GACA$ |
| 5 | ACA$ |
| 6 | CA$ |
| 7 | A$ |
| 8 | $ |

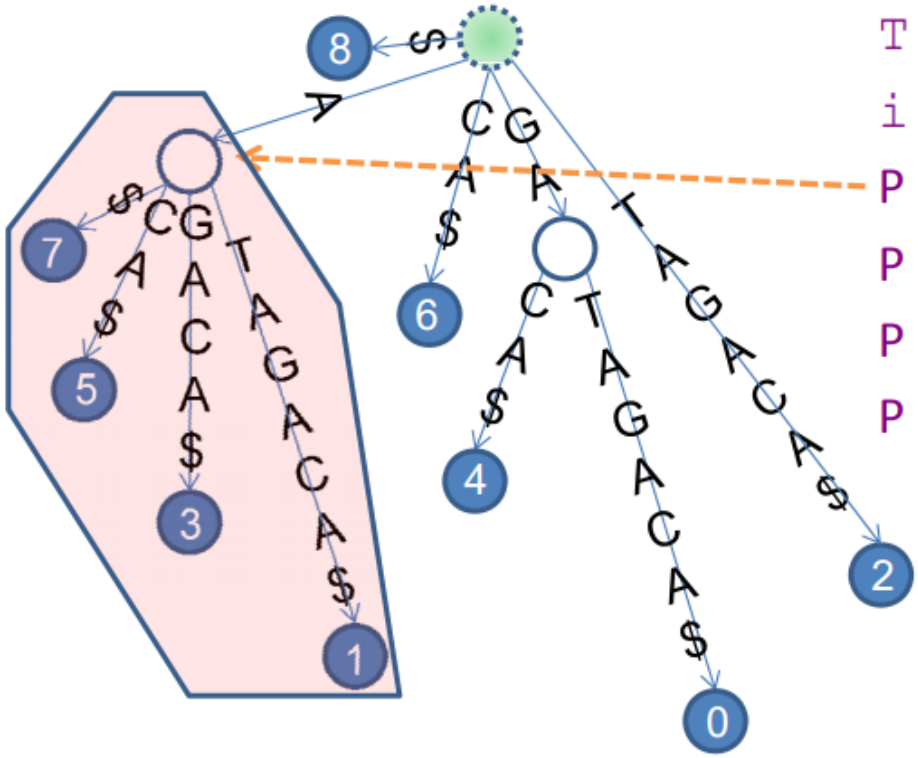



(example by Steven Halim)

# Suffix Tree (Search)



- To find all occurrences of **P** (of length m) in **T** (of length n)
  - Search for the vertex **x** in the Suffix Tree which represents **P**
  - All the leaves in the subtree rooted at x are the occurrences
- Time: O(**m** + occ) where occ is the total no. of occurrences




T = 'GATAGACA$'

i = '012345678'

P = 'A' → Occurrences: 7, 5, 3, 1

P = 'GA' → Occurrences: 4, 0

P = 'T' → Occurrences: 2

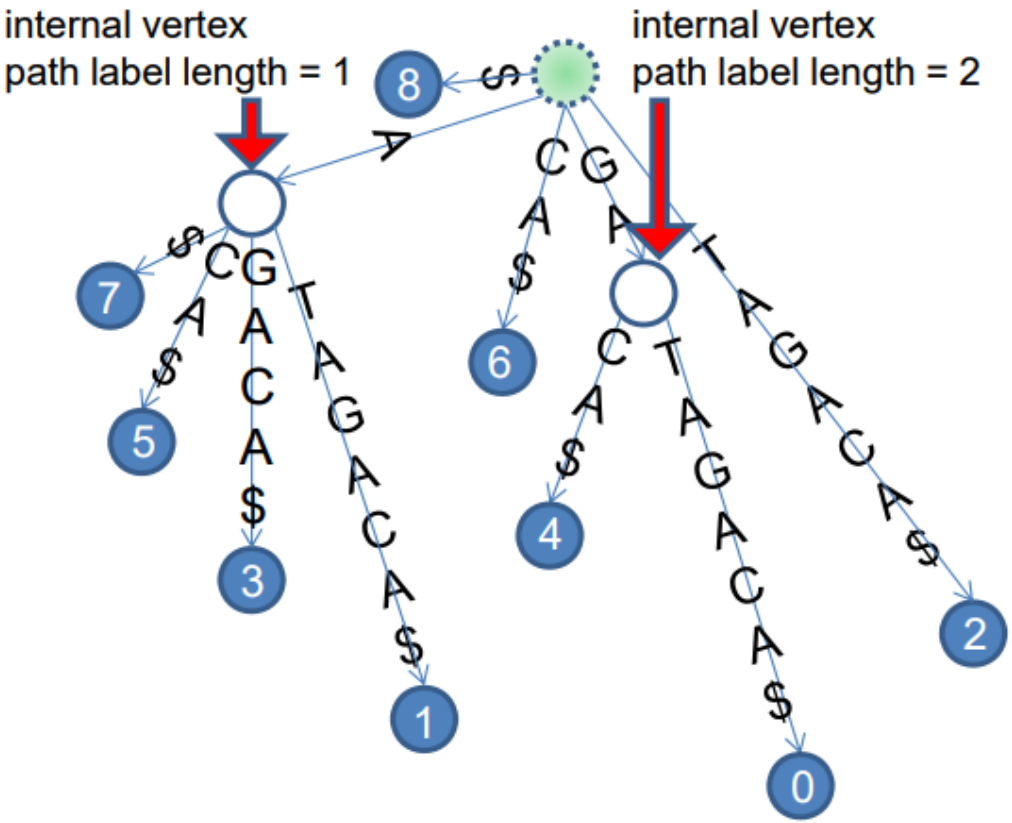
P = 'Z' → Not Found

(example by Steven Halim)

# Suffix Tree (Longest Repeated Substring)



- To find the longest repeated substring in T
  - Find the deepest internal node
- Time: O(n)




e.g. T = 'GATAGACA$'
The longest repeated substring is 'GA' with path label length = 2

The other repeated substring is 'A', but its path label length = 1

(example by Steven Halim)

# Suffix Tree (Longest Common Substring)



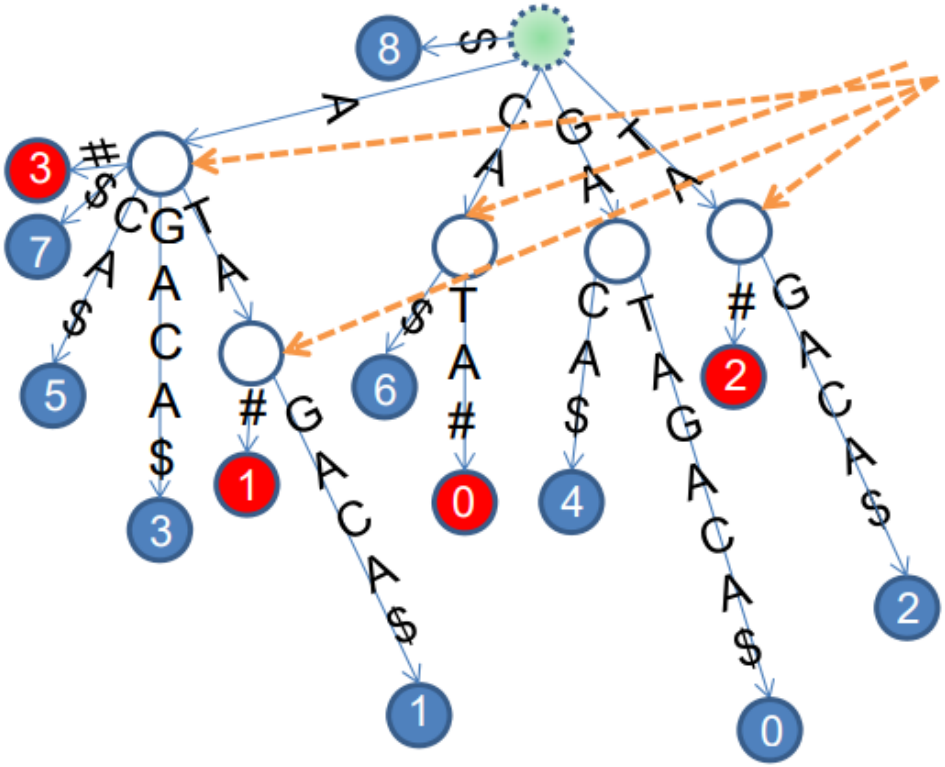
- To find the longest common substring of two or more strings
  - Note: In 1970, Donald Knuth conjectured that a linear time algorithm for this problem is impossible
  - Now, we know that it can be solved in linear time
  - E.g. consider two string T1 and T2,
    - Build a **generalized** Suffix Tree for T1 and T2
      - i.e. a Suffix Tree that combines both the Suffix Tree of T1 and T2
    - Mark internal vertices with leaves representing suffixes of both T1 and T2
    - Report the deepest marked vertex

(example by Steven Halim)

# Suffix Tree (Longest Common Substring)



- T1 = 'GATAGACA$' (end vertices labeled with blue)
  T2 = 'CATA#' (end vertices labeled with red)
  - Their longest common substring is 'ATA' with length 3




(example by Steven Halim)

# Suffix Tree (Construction)



See course webpage for references to constructing suffix trees.

(note: suffix ***tries*** are trivial to construct, while suffix ***trees*** are not)

# Suffix Array



The ***suffix array*** is a sorted array of all suffixes of a string.

Can do almost all things suffix trees can, and is also:

- … more space efficient;
- … easier to implement.

# Suffix Array



**Analogous to suffix sorting.**

A suffix **array** for ”GATAGACA$”:

| i | Suffix |
|---|---|
| 0 | GATAGACA$ |
| 1 | ATAGACA$ |
| 2 | TAGACA$ |
| 3 | AGACA$ |
| 4 | GACA$ |
| 5 | ACA$ |
| 6 | CA$ |
| 7 | A$ |
| 8 | $ |

Sort →

| i | SA[i] | Suffix |
|---|---|---|
| 0 | 8 | $ |
| 1 | 7 | A$ |
| 2 | 5 | ACA$ |
| 3 | 3 | AGACA$ |
| 4 | 1 | ATAGACA$ |
| 5 | 6 | CA$ |
| 6 | 4 | GACA$ |
| 7 | 0 | GATAGACA$ |
| 8 | 2 | TAGACA$ |

# Suffix Array (Construction)



## Naïve O(N² log N) implementation

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

char T[MAX_N]; int SA[MAX_N];

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; }

int main() {
   int n = (int)strlen(gets(T));
   for (int i = 0; i < n; i++) SA[i] = i;
   sort(SA, SA + n, cmp);
}
```

This is O(N)

What is the time complexity?
Can we do better?

Overall O(N² log N)

(example by Steven Halim)

# Suffix Array (Construction)



## $O(N \log^2 N)$ implementation

- An n-gram is an n-sized consecutive part of the original string. e.g. {***ab***, ***bb***, ***ba***} are the 2-grams of ***abba***, and ***abba*** is the only 4-gram.
- Algorithm: Sort by 2-grams (bigrams), 4-grams, 8-grams, etc. $\log_2(n)$ iterations needed.
- This sorting can be accomplished by using ***lexicographic renaming*** (i.e. using only bigrams and ranks), and the sorting comparator thus only needs to compare bigrams (ranks). Therefore, a comparison can be made in O(1) for all iterations.

# Suffix Array (Construction)



Example of ***lexicographic renaming***: Assume we have a sorted list of bigrams *{'ab', 'ab', 'ca', 'cd', 'cd', 'ea'}*. We then assign ranks by going from left to right, starting with rank 0 and incrementing the rank whenever we encounter a *changed* bigram:

```
ab : 0
ab : 0   [no change to previous]
ca : 1   [increment because different from previous]
cd : 2   [increment because different from previous]
cd : 2   [no change to previous]
ea : 3   [increment because different from previous]
```

Note that this would've been the first iteration in our suffix array construction algorithm.

(example by Johannes Goller)

# Suffix Array (Construction)



But what about 4-grams?

The naïve approach would involve comparing them character by character (i.e. 4 comparisons, and for n-grams, n comparisons).

Instead, we can look up the ranks of the two bigrams contained in them, and use the rank table we previously generated!

```
ab : 0
ab : 0   [no change to previous]
ca : 1   [increment because different from previous]
cd : 2   [increment because different from previous]
cd : 2   [no change to previous]
ea : 3   [increment because different from previous]
```

(example by Johannes Goller)

# Suffix Array (Construction)



## O(N log N) implementation

- Do the exact same thing, but with radix sort ***O(N)***.
  Note that the previous approach (i.e. quick sort) is sufficient for all problems in this course.

# Suffix Array (Construction)



**O(N) implementation**

- Ukkonen's algorithm (1995).
  Check it out if you want a challenge!

# Suffix Array (String Matching)



Finding all occurrences of P = 'GA' in S = 'GATAGACA'

Finding lower bound

| i | SA[i] | Suffix |
|---|---|---|
| 0 | 8 | $ |
| 1 | 7 | A$ |
| 2 | 5 | ACA$ |
| 3 | 3 | AGACA$ |
| 4 | 1 | ATAGACA$ |
| 5 | 6 | CA$ |
| 6 | 4 | GACA$ |
| 7 | 0 | GATAGACA$ |
| 8 | 2 | TAGACA$ |

Finding upper bound

| i | SA[i] | Suffix |
|---|---|---|
| 0 | 8 | $ |
| 1 | 7 | A$ |
| 2 | 5 | ACA$ |
| 3 | 3 | AGACA$ |
| 4 | 1 | ATAGACA$ |
| 5 | 6 | CA$ |
| 6 | 4 | GACA$ |
| 7 | 0 | GATAGACA$ |
| 8 | 2 | TAGACA$ |

$$2\ binary\ searches \Rightarrow \boldsymbol{O}(|\boldsymbol{P}|\log|\boldsymbol{S}|)$$

(example by Steven Halim)

# Suffix Array (Longest Repeated Substring)



- Simply find the highest entry in LCParray – $O(n)$

| i | SA[i] | LCP[i] | Suffix |
|---|---|---|---|
| 0 | 8 | 0 | $ |
| 1 | 7 | 0 | A$ |
| 2 | 5 | 1 | ACA$ |
| 3 | 3 | 1 | AGACA$ |
| 4 | 1 | 1 | ATAGACA$ |
| 5 | 6 | 0 | CA$ |
| 6 | 4 | 0 | GACA$ |
| **7** | **0** | **2** | **GATAGACA$** |
| 8 | 2 | 0 | TAGACA$ |

Recall:
LCP = Longest Common Prefix between two successive suffices

(example by Steven Halim)

# Suffix Array (Longest Common Substring)



- T1='GATAGACA$'
- T2='CATA#'
- T='GATAGACA$CATA#
- Find the highest number in LCP array provided that it comes from two suffices with different owner
  - Owner: Does this suffix belong to string 1 or string 2?
- *O(n)*

| i | SA[i] | LCP[i] | Owner | Suffix |
|---|---|---|---|---|
| 0 | 13 | 0 | 2 | # |
| 1 | 8 | 0 | 1 | $CATA# |
| 2 | 12 | 0 | 2 | A# |
| 3 | 7 | 1 | 1 | A$CATA# |
| 4 | 5 | 1 | 1 | ACA$CATA# |
| 5 | 3 | 1 | 1 | AGACA$CATA# |
| 6 | 10 | 1 | 2 | ATA# |
| **7** | **1** | **3** | **1** | **ATAGACA$CATA#** |
| 8 | 6 | 0 | 1 | CA$CATA# |
| 9 | 9 | 2 | 2 | CATA# |
| 10 | 4 | 0 | 1 | GACA$CATA# |
| 11 | 0 | 2 | 1 | GATAGACA$CATA# |
| 12 | 11 | 0 | 2 | TA# |
| 13 | 2 | 2 | 1 | TAGACA$CATA# |

# Summary



- Suffix Trie
- Suffix Tree
- Suffix Array